(12)

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 86-23-02	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) UW/NW VLSI Consortium Semiannual Technical Report No. 2		5. TYPE OF REPORT & PERIOD COVERED Technical, interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) UW/NW VLSI Consortium		8. CONTRACT OR GRANT NUMBER(s) MDA903-85-K-0072 ARPA-4563, #2 Code 5D30
9. PERFORMING ORGANIZATION NAME AND ADDRESS UW/NW VLSI Consortium, Dept. of Computer Science University of Washington, FR-35 Seattle, WA 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA - IPTO 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE March, 1986 13. NUMBER OF PAGES 40
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR University of Washington 315 University District Building 1107 NE 45th St., JD-16, Seattle, WA 98195		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

VLSI Design Generators, VLSI Consortium, Network C, CFL, Quarterhorse microprocessor, CMOS

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document reports on the research activities of the University of Washington/Northwest VLSI Consortium for the period October 1, 1985 to March 17, 1986 under sponsorship of the Defense Advanced Research Projects Agency, under contract number MDA903-85-K-0072, program code number 5D30.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UW/NW VLSI CONSORTIUM

Semiannual Technical Report No. 2

University of Washington

March 17, 1986

Reporting Period: 1 October 1985 to 17 March 1986

Principal Investigator: Lawrence Snyder

Sponsored by
Defense Advanced Research Projects Agency(Dod)
ARPA Order No. 4563/2
Issued by Defense Supply Service-Washington
Under Contract #MDA903-85-K-0072
(Program Code Number: 5D30)

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Contents

1 Executive Summary

1.1 Scope of This Report

This document reports on the research activities of the UW / NW VLSI Consortium for the period 1 October 1985 to 17 March 1986 under sponsorship of the Defense Advanced Research Projects Agency. The applicable contract for this period is MDA903-85-K-0072.

1.2 Accomplishments

During this period, the Consortium staff refined a number of the design generators reported on in the last Technical Report. Two substantial design efforts, a 32 bit microprocessor and a digital filter, utilized several of the generators. Ongoing work focuses on development of interfaces to verification tools such as the DRC and switch level simulators, as well as a functional simulator currently under development. Such interfaces allow the verification tools to make use of the correctness of instances created by generators.

A preliminary model for generator construction has been proposed. The intent of the model is to provide a concise specificiation of the circuit from which a number of output descriptions such as the layout, schematic and transistor netlist may be derived. The model has been applied to both a decoder and a multiplier generator.

Work is progressing on a simulation system intended to provide a broad range of capability – from high level behavioral modeling to low level transistor modeling. The system is intended to be used to simulate assemblies of circuits produced by the design generators as well as hand crafted circuits.

The Consortium supported University instruction in VLSI design through use of Consortium hardware and software design tools. The staff is currently preparing for an intensive class in CMOS design, oriented towards industry engineers.

Since June 1985 the Consortium has distributed Release 3.0 of its toolset to over 90 sites. Future releases will include the suite of CMOS design generators described in the last technical report.

2 Progress in Design Generators

2.1 Development of a Model for Generator Construction

During the last six months, a model for generator construction has been developed by Principal Investigator Larry Snyder, Professor Jean-Loup Baer, and graduate student Meei-Chiueh Liem. Based on a descriptive static language, the model describes at an abstract level the layout, schematic and transistor netlist of the circuit being generated. The intent here is to employ a single model at a sufficiently abstract level that it specifies the essential common elements of all three descriptions. At a lower level of detail these descriptions will of course differ and require different means of specification.

An initial version of this language has been formulated and applied to several generators, including a decoder and multiplier. The success of a model such as this depends upon its range of applicability. Thus our current effort is to apply the language to a number of generators in order to determine the adequacy of its descriptive elements. Once this is accomplished, the details of producing specific descriptions (e.g. layout, schematic etc.) from the model may be addressed.

2.2 Interfaces to Verification Tools

Generators are useful to designers simply because they produce subcircuits that can be combined to make a complex IC. The task of assembling the generated subcircuits and verifiying the resulting IC can also be made more efficient by interfacing the generator outputs to the verification tools.

The DRC task is one example. By definition the layouts produced by generators are design rule correct. By making use of this fact, the DRC can be made considerably more efficient. The type of interface required may be dependent on the methods employed by the DRC. A hierarchical DRC such as the one in MAGIC employs timestamps to avoid repeated checking of blocks of layout. If the generator employs the same timestamping convention, then MAGIC's DRC will be saved the effort of checking most of the generated block. Our generators currently support this timestamping convention when generating MAGIC-formatted layouts.

Another type of checker is one that operates on a flattened design. The Carnegie-Mellon DRC is an example of this variety. An interface in this case is a border

description that contains all of the layout within a maximum design rule of the bounding box of the generated block (assuming of course that other circuitry does not overlap this bounding box). One of our graduate students, Mary Bailey, has written a program which produces this structure. We are currently evaluating the improvement in efficiency obtained by substituting this ring structure for the entire layout.

Another example of an interface is the transistor netlist used by switch level simulators such as RNL. The traditional method of laying out an entire circuit and extracting the transistor netlist can be improved upon considerably. By extracting generated blocks individually and assembling the resulting netlists, one can alter one part of a design and reconstruct the entire netlist with a minimum of effort. Hierarchical extractors such as that used in MAGIC support this interface. By the use of netlist merging utilities, flat extractors like MEXTRA can obtain a similar savings of effort. One of our graduate students, Robert Cypher, has written a general netlist merging utility for netlists derived from MEXTRA or the MIT utility NETLIST.

2.3 Applications of Design Generators

The existing set of generators has been used in the design of several IC's of > 20K transistors. The Quarter Horse, a 32 bit microprocessor, was described in the previous DARPA technical report. The Quarter Horse employed generators for a PLA, register file, and padframe. Ongoing modifications to the Quarter Horse will likely employ generators for a multiplier and ROM (see section 4).

Another complex design to utilize generators is a variable order IIR digital filter designed by Hyong Lee, a graduate student supported by the Consortium (see Appendix III). The design implemented a filter of up to eighth order for the purpose of speech processing. A PLA, multiplier, and padframe were all generated, the remainder of the circuitry being custom designed. An effort is currently underway to design an FIR filter entirely with the use of generators.

Designs such as the Quarter Horse and digital filter provide diverse applications to test the usefulness of the design generators and their interfaces to the verification tools.

2.4 Quality Assurance of Generated Parts

One of the main goals of the project is to develop a methodology for insuring that generators produce quality designs over their entire range of parameters. For a number of generators we have developed comprehensive test suites that verify the design rule correctness as well as the functionality of instances. By automating this procedure, we have simplified the verification process required whenever a design revision is performed. These test suites have already proven useful in an update of the multiplier generator.

Work is also proceeding to characterize the performance of generated parts. The recently purchased Northwest Instruments Test System has provided us with considerable timing data on the chips fabricated by MOSIS. To date we have tested instances of a multiplier, ROM, counter, PLA and shifter. Information provided by the tests has allowed us to verify the models employed by CRYSTAL and RNL.

3 Progress on the Network C Simulation System

Network C is a multipurpose simulation system currently being developed by Bill Beckett of the Consortium staff. Network C, or NC, is described in detail in Appendix II. When complete, NC will simulate circuits composed of high level functional models as well as MOS transistor models. Development work was initiated on the UW Academic Computer Center's CDC CYBER, and is continuing on the Consortium VAX.

At this point, the high level functional capabilities of NC are working on the VAX. These capabilities include the hierarchical specification of networks of functional models and specification of the procedural bodies of these models. The system at this level of development can be thought of as a discrete event simulator imbedded in C. Both integer and floating types for network node values are implemented. A production version of the system has been installed for use by the Consortium staff. A FIR filter description with loadable coefficients has been successfully modeled using Network C.

In addition to the NC translator and run time system now installed, a new event file plot program has been written to be used with NC. This program, called SCP, is considerably faster than a previous plotting program since it does not require the event list to be sorted by signal name.

The developmental version of the system on the CYBER includes both the piecewise linear MOS circuit simulation technique described in Appendix II and a general non-linear system solver for strictly Kirchoff simulations. The MOS circuit simulation modules have been translated to C and ported to the VAX. Currently, they have not been integrated into the system. The non-linear equation solver has not been translated to C yet.

The work currently in progress on the VAX version is the restructuring of the translation scheme around a more dynamic definition of the parse tree node. The new node definition allows a simplification of the rule transformers since the type of each node is no longer a compile-time specification. It was found that the translation technique using the former scheme was becoming unwieldy as the number of rule transformers began to grow.

The current short term plan is to complete the restructuring of the rule transformers and integrate the MOS simulation modules. Implementation of these modules requires that the syntax be extended to include the type *mos*, the notation for derivatives, and the notation for subfields of the node records associated with network variables.

When the MOS simulation is working, a new production version will be installed and at the same time a reference manual will be ready.

4 Update on the Quarter Horse Microprocessor

4.1 Modifications to the Design

The Quarter Horse is a 32 bit microprocessor designed by a graduate level VLSI class at the University of Washington during winter quarter 1985 (A description is appended to the last technical report). Since that time a number of modifications have been made and the result termed QH2. The datapath has been rotated to fit the short dimension of a 7200 by 9200 micron die. This allows approximately one-third of the die to be used for various extensions and modifications to the base QH2 architecture. Extensions currently in progress are a multiplier, a ROM for storing microcode and variable register windows.

In addition, the QH2 has a completely redesigned program counter, shifter and ALU. A new PLA controller is based on a generator developed by the Consortium staff. The global floorplan and routing was produced by Coordinate Free LAP

(described in Appendix I), permitting greater flexibility in the size and design of the parts of the processor. The QH2 also features a LSSD register on the outputs of the PLA, allowing greater ability to test the chip after fabrication.

The database of the QH2 was converted from the Caesar structure to the MAGIC format, allowing use of the MAGIC graphical editor and its interactive design rule checker. After conversion from MAGIC format to the CIF standard, we used CDRC (based on the design rule checker written at CMU) for batch design rule checking at the global level. MEXTRA was employed for circuit extraction preparatory to simulation. Switch-level simulation of the chip was done with RNL, the test vectors being generated by a C program written for that purpose.

4.2 Design of the Quarter Horse Test PCB

The Quarter Horse Test Board is a four-layer Multibus printed circuit board designed with the purpose of testing the Quarter Horse microprocessor. The board is 6.75 by 12.00 inches in size and contains a socket for the Quarter Horse, 64K words of system memory (a word is 32 bits), a 15MHz two-phase programmable system clock and interrupt logic. Test pins are also provided to allow for easy observation of the Quarter Horse signals by a logic state analyzer. Design of the PCB was an independent study project of undergraduate Diane Honda.

There are two modes of operation of the Quarter Horse Test Board: the Multibus mode and the Quarter Horse mode. In the Multibus mode the board acts as a slave to the host computer system. In the Quarter Horse mode, external requests cause the Quarter Horse microprocessor to execute programs resident in the system memory. The intention is to load a program into system memory and then have the Quarter Horse run it. Upon completion the program memory may be examined via the Multibus interface.

In designing the board the major considerations were to have a fast system clock (the initial specification called for a 20MHz system clock) and to choose a memory fast enough so as not to limit the Quarter Horse reads/writes. Speed of reads/writes between the host computer and the system memory was not a consideration. An attempt was also made to have some flexibility on the board such as a programmable clock. Other design issues taken into account were board space, availability and expense of parts and ease of routing.

When the board design was completed a printed circuit board editor called PCB was

used to do the board layout. Written by Andreas Nowatzyk of CMU, PCB supports four-layer rectangular boards and generates a CIF output file that complies with the design rules specified by MOSIS for four-layer boards. Unfortunately a Multibus board is not rectangular so additional CIF was written which would make the board the desired shape. The CIF output file was then edited using a text-editor. This edited CIF file was then sent to MOSIS for printed circuit board fabrication.

5 Distribution of the Consortium Toolset

In June, 1985 the Consortium began distributing Release 3.0 of the VLSI design toolset. Currently 90 sites have received this release, which supports MOSIS nMOS as well as CMOS processes. Feedback forms returned by 18 of these sites indicate that at least 120 designs have been constructed with Release 3.0. A distribution planned for summer, 1986 will contain a number of the generators developed under DARPA funding.

An effort is underway to port the toolset from our 4.2 BSD VAX to an Apollo DN550 node running Domain IX. The effort is proceeding smoothly with about two thirds of the effort completed. Many of the tools have also been ported to a RIDGE-32.

6 Intensive Class in CMOS Design

During April and May the Consortium will offer its fourth intensive class in CMOS design. The course provides the background and lab experience for designing digital CMOS circuits. Participants have the option of implementing a 3 micron CMOS design of their own choice during the course. The design will be fabricated through the MOSIS commercial facility and the Consortium test facility will be available to the participants for testing the returned parts.

Appendix I

Coordinate Free LAP

This paper was submitted to the 1986 Design Automation Conference.

William Beckett
UW/NW VLSI Consortium
Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

Coordinate Free LAP

Abstract

Coordinate Free LAP (CFL) is a library of subroutines written in C intended to facilitate the construction of VLSI circuit layouts. The operators of CFL generate new cells by forming combinations of existing cells using only relative positioning, that is without reference to a system of coordinates. The external data representation used by CFL may be made compatible with either of the UCB graphics editors, Caesar and Magic, so these editors may be used in conjunction with CFL. CFL is able to assemble sets of large cells very quickly because its positioning and routing operations work from descriptions of the boundaries of cells and, therefore, avoid direct references to the geometry within cells.

This paper bears on topics 3 (IC Layout) and 4 (Silicon Compilation).

Coordinate Free LAP

Introduction

Coordinate Free LAP (CFL) is a library of subroutines written in C intended to facilitate the construction of VLSI circuit layouts. The system is organized algebraically in that there is a data type called SYMBOL, a set of operands of this type, and a set of operators which generate new SYMBOLs by forming combinations of existing SYMBOLs.

The system has only two geometric primitives, **box** and **label**, which may combined to make objects. There is a larger set of non-primitive objects called macros, which may be used to generate frequently used structures such as contacts. Routing facilities are provided which generate a variety of planar and non-planar wiring patterns used to connect functional blocks. Additionally, there is a coordinate dependent facility called **wire** for generating arbitrary configurations of material.

Although CFL has sufficient functionality to allow definitions to be developed for all artwork including the lower level cells in a design, it is intended to be used more in the mode of chip assembly. Hence the typical application involves using a graphics editor to generate lower level cells or tiles and then using CFL facilities to assemble these leaf cells into higher level modules. Currently, the system may be used in conjunction with either of the UCB graphics editors, Caesar[3] and Magic[4].

To insure that a wide variety of assembly situations can be accommodated, CFL includes approximately 70 variants of operators for juxtaposing, transforming, and replicating hierarchies of symbols. The positioning performed by most of these operators is with respect to several abstract locations associated with objects, for example, 'the top', rather than to a set of coordinates.

The syntax of these operators is quite compact since generated symbols are simply stored in program variables of type SYMBOL *. The embedding of the language in C is such that sequences of CFL operators admit to both procedural and declarative interpretations. The resulting coordinate free form for defining the structure of complex objects is grammatical in character and fairly easy to manipulate.

All of the calculations which support the operators of CFL are performed from descriptions of the borders of the symbols. The information in the border descriptions includes the bounding box and lists of rectangles representing the intersection that each kind of material in the symbol makes with the bounding box. If there is a label near this intersection, the border description will also contain the label. If a border description is available for a particular symbol, CFL will not require access to any of the rest of the geometry of symbol.

The system will automatically generate border descriptions from the geometry whenever the need arises but it will also automatically save them on disk when library symbols are written out. In this way, modules which have a large number of rectangles may be

accessed from the library without the need of reading all of the geometry files associated with their sub-modules. This capability allows CFL to assemble large blocks of circuitry extremely quickly.

CFL provides automatic hierarchy compression when symbols are written to disk so that only those symbols which represent meaningful functional groups need be saved.

Entities

The operations provided by CFL are defined with respect to a number of basic entities. These entities include primitive geometric objects and compound objects, called SYMBOLS; the boundaries of these symbols, called BORDERS; and individual symbolic points along these boundaries.

CFL has the following two primitive objects -

box(*layer,dx,dy*) - box
label(*name,dx,dy,pos*) - rectangular label

These are the same two primitives used by Caesar and Magic. (In the case of Magic, the label primitive also specifies a layer.) All coordinates are dimensionless. **box** creates a box on the specified layer with dimensions *dx* and *dy*. **label** creates a label. Labels consist of a rectangle with dimensions *dx* and *dy* and a *name*. *pos* is used to specify the position of the name of the label relative to its center when it is displayed by the graphics editor.

A CFL symbol is either a primitive object or an object formed by combining primitive objects and other symbols using the CFL operators. Each symbol is a collection of geometry (boxes), calls to other symbols (calls) and labels. CFL represents symbols internally as data structures having lists of boxes, calls, and labels and all references to symbols within a CFL application program are made through pointers to these structures. The pointers are declared with the declarator SYMBOL *.

For example,

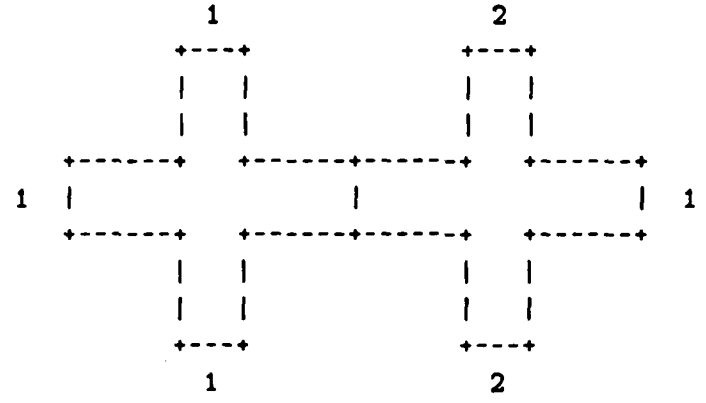
```
SYMBOL *box1,*box2,*cross1,*pair1;
box1 = box("metal", 3,10);      /* vertical bar   */
box2 = box("metal",10, 3);      /* horizontal bar */
cross1 = cc(box1,box2);         /* metal cross    */
pair1  = cx(cross1,cross1);     /* two adjacent crosses */
```

In this example, **cc** is the center to center alignment operator of CFL. It creates a new object by juxtaposing the center of the vertical bar and the center of the horizontal bar. The operator **cx** constructs a horizontal pair of crosses, aligned by their horizontal center lines, with the right edge of the first cross abutting the left edge of the second cross. (All

CFL operators are declared SYMBOL * by the include file *cfl.h* which must be included in CFL application programs.)

For each symbol, CFL maintains a list of coordinates which mark the centers of all intersections of mask layers and the bounding box. These sets of coordinates, called crossings, are maintained separately for each mask layer and for each of the four sides of the bounding box. Each crossing may be referred to by specifying its symbol, side of the bounding box, layer and ordinal along the side.

For example, the symbol, *pair1*, generated above looks some thing like this -



The crossings are given by the following four-tuples -

```
(pair1, "top",   "metal", 1)
(pair1, "top",   "metal", 2)
(pair1, "bot",   "metal", 1)
(pair1, "bot",   "metal", 2)
(pair1, "left",  "metal", 1)
(pair1, "right", "metal", 1)
```

The string literals "top", "bot", "left", and "right" are used by CFL to indicate the sides of bounding boxes. Layer names like "metal" are, of course, technology dependent. For each technology, CFL uses the long format Caesar or Magic layer names. All crossing ordinals start at 1 and increase along the coordinate corresponding to the bounding box edge in question.

Several of the routing operators in CFL have symbolic points as arguments. These arguments are declared to be of type PT * and are generated by the symbolic point descriptor constructor, **pt**. For example, to construct symbolic points which refer to the leftmost and rightmost metal crossings in *pair1* above, the following program statements are used -

```
PT *p1,*p2;
p1 = pt(pair1, "left",  "metal", 1);
p2 = pt(pair1, "right", "metal", 1);
```

Whereas symbolic points are used to refer to specific crossings, CFL borders are used to refer to sets of crossings. CFL borders are used as arguments to some of the routers. A border is similar to a symbolic point in that it is referenced through a descriptor, declared BORDER *, and constructed using a constructor, in this case, **bd**. In the simplest case, a border contains all the crossings associated with a given symbol, side and layer. Hence,

```
BORDER *b1,*b2;
b1 = bd(pair1, "top", "metal");
b2 = bd(pair1, "bot", "metal");
```

constructs two borders; *b1*, containing all the metal crossings on the top of *pair1*, and *b2*, containing all the metal crossings on the bottom of *pair1*.

In addition to the basic border constructor which, by default, includes all crossings in its resulting border description, CFL provides operators **bdin** and **bdex** for including and excluding specific crossing ordinals from border descriptions. In general then, the border description facilities are capable of directing the routers to consider any subset of crossings along the side of a particular symbol. For example, the following statements construct a description of the top of pair1 which includes only the second crossing:

```
b1 = bd(pair1, "top", "metal");
b1 = bdex(b1,1);
```

In the special instance that the ordinal argument is zero, **bdin** will include all crossings in its resulting border and **bdex** will exclude all crossings from its resulting border.

Operators

CFL has six classes of operators -

1. Alignment operators
2. Linear transformations
3. Array constructors
4. Tiling operators
5. Library access operators
6. Miscellaneous operators

The alignment operators combine a pair of symbols by placing them in one of several relationships with respect to each other. The coordinate free nature of CFL stems largely from the fact that the alignment operators typically specify the position of one symbol relative to another rather than the position of either of them relative to a more global set

of coordinates. CFL has six categories of alignment implemented as the following thirteen alignment operators -

1. Center to center	**cc**
2. Center line to center line	**cx,cy**
3. Edge to edge	**ll,rr, tt,bb**
4. Border to border	**bx,by**
5. Point to point or center	**pax,pay, cp**
6. Origin to origin	**oo**

Each of these operators has two arguments *s1* and *s2* which are symbol pointers, declared SYMBOL *. The operators form a new symbol containing *s1* and *s2* positioned according to the indicated alignment criterion. The position of *s2* relative to *s1* in this new symbol is called the (0,0) position. All of the alignment operators have three additional variations which allow the specification of offsets from this (0,0) position in the *x*, *y*, or both directions. The variations are formed by suffixing the operator name with **dx**, **dy**, or **dxy**. For example, the **cx** operator has the following four forms:

cx(*s1,s2*)	- pair in x, center lines aligned
cxdx(*s1,s2,dx*)	- pair in x, center lines aligned, x offset
cxdy(*s1,s2,dy*)	- pair in x, center lines aligned, y offset
cxdxy(*s1,s2,dx,dy*)	- pair in x, center lines aligned, xy offset

The center to center, center line to center line, and edge to edge alignment operators depend only on the bounding boxes of the symbols being aligned. The border and point alignment operators .however, depend on the border crossings. For example, the **bx** operator forms the horizontal pair of symbols (*s1,s2*) such that the right side of the bounding box of *s1* is adjacent to the left side of the bounding box of *s2* and the symbols are aligned so that corresponding patterns of material along the common edge match up.

The point alignment operators are similar to the border alignment operators except that the symbols are aligned so that specfic symbolic points along the respective borders are adjacent.

The origin to origin operator is used in conjunction with CFL's routers and will be discussed later.

There are three linear transformations -

mx(*s*)	- mirror in x
my(*s*)	- mirror in y
rot(*s,n*)	- rotate

The argument to **rot** is in degrees and must be an integer multiple of 90.

There are three array constructors, **nx**, **ny**, and **nxy**, which can be used to generate horizontal, vertical, or rectangular arrays of a given symbol. As in the case of the alignment operators, offset variants of these operators are also defined. The interpretation of the offsets is, however, slightly different. *dx* and *dy*, when supplied, are taken to be the spacings between the bounding boxes of successive array elements. The (0,0) position is when the bounding boxes are adjacent. The variants of the array operators which would produce a non-rectangular structure are not defined, for example, **nxdy**.

nx(*s,n*)	- repeat in x
nxy(*s,nx,ny*)	- repeat in x and y
ny(*s,n*)	- repeat in y

There are three additional array constructors **repx**, **repy** and **repxy** which construct arrays of particular spatial periods. The arguments to these routines are given as *dx* and *dy* but they specify the periods rather than offsets. These operators do not have variants for providing additional offsets.

repx(*s,n,dx*)	- repeat in x with period dx
repxy(*s,nx,ny,dx,dy*)	- repeat in x and y, with periods dx dy
repy(*s,n,dy*)	- repeat in y with period dy

Tiling is similar to an array operation except that each element of the generated array can be a different symbol. There are three tiling operators, **vx**, **vy**, and **vxy**, which can be used to generate horizontal, vertical, or rectangular tilings. These operators are similar to the array operators except that the first argument is an array of symbol pointers rather than a single symbol pointer. The tiling operators, then, operate on vectors of symbols so their mnemonic starts with **v**. There are no offset variants for the tiling operators since the offset for each tile could potentially be different.

vx(*s,n*)	- vector in x
vxy(*s,nx,ny*)	- vector in x and y
vy(*s,n*)	- vector in y

There are two operators for accessing library symbols -

gs(*cell*)	- get library symbol
ps(*y,s*)	- put symbol in the symbol table

gs will read a library symbol in either Caesar or Magic format, place the symbol in the data base and return a pointer to it. If the symbol is already in the data base, **gs** simply returns the pointer, that is, it will read the symbol only once.

ps compresses the hierarchy below its argument and marks that argument as a library symbol. The hierarchy compressor removes from the hierarchy all cells which are not

marked as library cells, that is, cells which were not read in with **gs** or cells which have not been marked as permanent by a call to **ps**. Therefore, **ps** can be used to not only to save symbols but also to control the actual structure of the hierarchy.

CFL is designed to be able to be used with any desired technology. It obtains its table of layer names from technology files in the CFL path. A call to the routine **cflstart** initializes the package and specifies the name of the technology file to be used. **cflstart** must be called before invoking any other CFL functions. **cflstop** causes all permanent symbols to be written to disk and should be called just prior to exiting a CFL application.

Routers

CFL does not currently provide high level routing facilities such as a general channel router or switchbox router. Rather, the CFL routers consist of a set of wiring pattern generators each of which is specialized to a particular kind of routing situation. These routers, which are designed to be used in conjunction with each other and the other CFL operators, support a set of elementary routing operations from which more sophisticated patterns may be constructed.

There are two types of routing facilities available in CFL, planar routers and non-planar routers. The planar routers are -

pp(*s0,p1,p2,w*)	- point to point router
pr(*s0,b1,b2,w*)	- general planar router
ext(*b,d,w*)	- border extender
fill(*s,side,d*)	- Caesar fill operation

and the non-planar routers are -

plx(*s0,p1,p2,w,ct*)	- horizontal point to line router
ply(*s0,p1,p2,w,ct*)	- vertical point to line router
elb(*s0,b1,b2,w,ct,rev*)	- general elbow
tee(*s0,b1,b2,w,ct,rev*)	- tee

Since CFL is coordinate free, the routers operate from border descriptions and from symbolic point designations. The generation of symbolic point and border descriptors is described in the earlier section, Entities.

Most CFL operators produce a new symbol by combining existing symbols. The arguments to these operators have no particular spatial relationship to each other before the operation takes place. The routers, on the other hand, rather than combining symbols, must form connections between them. This process requires that the symbols to be connected have a previously established fixed spatial relationship.

Symbols acquire a fixed spatial relationship as soon as they become constituents of some higher level symbol. CFL refers to a higher level symbol, *s0*, which contains symbols *s1* and *s2* as a container of *s1* and *s2*. Within any container, the relative positions of *s1* and *s2* are fixed.

The routers, like all other CFL operators, are SYMBOL * valued functions. When a router is invoked it produces a pattern of wiring as its result. This pattern of wiring is not 'written' into place directly by the routing operation, rather it is a separate symbol in its own right. Therefore to connect two symbols using the routers, two steps are necessary:

1. Use one of the routers to generate the pattern of wiring necessary to form the required connections.
2. Use the origin to origin alignment operator to locate the generated wiring pattern in the container so that the intended connections are made.

In all cases, the wiring is generated in the coordinate system of the container and often the two steps above may be combined using a statement of the following form -

$$result = \mathbf{oo}(container, \mathbf{router}(container, ...));$$

The rationale for requiring that the generation and placement of wiring patterns be performed in steps rather than as an atomic operation is that in many cases routing problems require the generation of complex patterns in which wiring generated by one call to a router must itself be connected to the wiring generated by another call to a router. Separating the generation allows the generated wiring to become a separate symbol which may be then operated on using any CFL operator.

The point to point router generates a single wire for connecting two symbolic points (see Entities). The layers of the points should match and both points must be uniquely locatable within the containing symbol.

The planar router generates a planar wiring pattern for connecting the points in two borders (see Entities). The borders must contain the same number of points. Also they must be uniquely locatable within the containing symbol. All wires will have the same width.

To simplify the diagnostic process, **pr** will construct wiring patterns whether or not there is sufficient space for the number of wires requested. It will, however, issue a warning message if any of the generated wires are closer than a specified tolerance.

ext generates a pattern of wiring for extending all points in a given border perpendicularly for a specified distance. **fill** is similar to **ext** except that all layers crossing the indicated side are extended. The extensions have the same widths as the crossings. For example, suppose it is desired to generate a symbol *s2* which consists of five instances of a symbol *s1* placed a distance 10 apart and connected by extending the material of the right side of *s1*. The following CFL statement generates this configuration -

```
s2 = cx(nx(oo(s1,fill(s1,"right",10)),4),s1);
```

Generally speaking the planar routing facilities of CFL are technology independent whereas the non-planar routing facilities are technnology dependent since contacts must be specified.

plx generates a single wire for connecting a symbolic point the vertical line running through another symbolic point. The connection is made horizontally. The layer of the first point taken to be the layer of the wire. The points must be uniquely locatable in the container symbol. If requested, a contact is placed with its origin at the intersection of the vertical line and the generated wire.

ply is similar to **plx** but generates a single wire for connecting a symbolic point to the horizontal line running through another symbolic point. The connection is made vertically.

elb generates a wiring pattern for connecting the points in two borders, say, *b1* and *b2*. Wires from *b1* and *b2* may have different widths. The pattern generated must form an elbow but it is not necessary that *b1* and *b2* be on the same layer. If requested, a contact will be placed with its origin at the intersections of the wires from *b1* and the wires from *b2*.

elb may generate either forward or reversed elbows. For a forward elbow the low order points in *b1* will connect to low order points in *b2*. For a reversed elbow, low order points in *b1* will connect to high order points in *b2*.

Through combinations of selecting subsets of the borders with **bdin** and **bdex** and utilizing the normal and reverse options, a succession of **elb** invocations may be used to form a set of elbows between *b1* and *b2* which implement any desired ordering of the connections.

tee generates a wiring pattern for connecting the border of a tee connected symbol to the wiring of a transverse routing symbol. The wiring in the routing symbol is assumed to run perpendicular to the wiring generated for connecting the tee connected symbol. The routing symbol, presumably generated by a prior call to a router, is also assumed to consist strictly of parallel lines, no elbows. All generated wires will have the same width. If requested, a contact will be placed with its origin at the intersection of the generated wires and the wires existing in the routing symbol. The connection order for tee may be either forward or reversed.

All of the non-planar routers have a contact argument *ct*. The provision for positioning this contact in generated routing is coordinate dependent in that the contacts are always positioned so that their origins, coordinate (0.0), coincides with the intersections of wires on different layers. If the contacts are symmetric and generated with the CFL box primitive, as is the case with the NMOS macros **gb** and **rb**, the origins will be in the geometric centers because the box primitive is designed to make boxes which are symmetric about the origin whenever possible. If other, asymmetric, forms of contacts are needed they may be generated according to the above criterion using the CFL wire facility described later.

Use of the routers generally requires that three pointers into a symbol hierarchy be supplied - the container and the two symbols to be connected. When symbols are retrieved

from the library using **gs** only one pointer is provided. A typical problem of this form is to retrieve from the library both a complete circuit and a pad frame and then to connect the circuit to the pads. The CFL procedure **locate** may be used to obtain a pointer to any named sub-symbol within a symbol hierarchy. All symbols saved with **ps** are named symbols.

For example, suppose a circuit called *memory* is to be placed in a pad frame and connected. Suppose further that the section of the pad frame containing the output pads is named *outputs* and that the memory outputs are available on the boundary of a sub-symbol called *planes*. The following CFL code accomplishes the task:

```
SYMBOL *memory,*pads,
       *outputs,*planes,*chip;

/* get the memory and the pad frame from the library          */

memory = gs("memory");
pads   = gs("pads");

/* establish pointers to the planes sub-symbol of the memory */
/* and the outputs sub-symbol of the pad frame                */

outputs = locate(pads,"outputs");
planes  = locate(memory,"planes");

/* position the memory within the padframe                    */

chip = ccdx(memory,pads,120);

/* connect the outputs from the memory planes to the          */
/* corresponding output pads                                  */

chip = oo(chip,pr(chip,bd(planes, "top","metal"),
                       bd(outputs,"bot","metal"),3));
```

Macros

CFL has two groups of macros - technology independent macros and technology dependent macros. The technology independent macros are -

alpha(*s,layer,w*)	- character string, width w
cross(*layer1,dx1,dy1,layer2,dx2,dy2*)	- two boxes, centers aligned
letter(*c,layer,w*)	- alphanumeric letter, width w
lne(*layer,w,dx,dy*)	- el, north east
lnw(*layer,w,dx,dy*)	- el, north west
lse(*layer,w,dx,dy*)	- el, south east
lsw(*layer,w,dx,dy*)	- el, south west

alpha generates a string of characters which are 5w wide, 8w high with 2w spacing in between. The same rules apply to **letter**. The character set that is available is

A - Z
0 - 9
- . , : ; ? ! / [ ] + =

Currently, space (or blank) is not available.

The technology dependent macros available generate commonly used structures like contacts, pullups and and components of standard pad frames.

Wire Facility

In order to provide for the parametric generation of particularly complex leaf cells, or cells with specific coordinate requirements like router contacts, CFL includes the wire facility which allows the use of symbol relative coordinates. Note that the use of this facility can introduce significant coordinate dependency into a design so it should not in general be used in cases where the coordinate independent operators are able to serve. The procedures associated with the wire facility are the following -

wire(*layer,width*)	- Initialize a wire
at(*x0,y0*)	- Move to the point (x0,y0)
dx(*dx0*)	- Draw to the point (x+dx0,y)
dy(*dy0*)	- Draw to the point (x,y+dy0)
iso(*s*)	- Include symbol origin
wl(*layer*)	- Reset the wire layer
ww(*width*)	- Reset the wire width
x(*x0*)	- Draw to the point (x0,y)
y(*y0*)	- Draw to the point (x,y0)

wire is of type SYMBOL *. All of the procedures apply to the wire generated by the last call to **wire**. Note that the symbol generated by **wire** may contain an arbitrary number of physical 'wires' which need not be connected. The only thing they have in common is their coordinate system.

The procedure **iso** has a symbol as its argument. **iso** includes that symbol positioned so that its origin coincides with the current wire position. Note that the current wire position, or more precisely, the position within the coordinate system of the current wire, is initialized with the at procedure and maintained by all move and draw procedures.

Experience Using CFL

The UW/NW VLSI Consortium has been using CFL for the last year to make a set of module generators. These generators are designed to produce instances of general structures which meet various specifications. For example, a CMOS multiplier generator has been developed which produces two's complement multipliers for either signed or unsigned operands of varying sizes. Flexible generators have also been developed for a PLA, a CAM, several kinds of ROM's and a multiplexer. In general, the generators include features like automatic adjustment of driver and buss sizes as a function of the modules' speed and power requirements.

In addition to the module generators, CFL has been used for assembling and routing the components of the Quarter Horse microprocessor that the Consortium has developed.

So far indications are that CFL is easily learned by those familiar with C. The number and sohpistication of the projects that have been completed using CFL indicate that the system is substantially more convenient than coordinate based systems while still retaining a sufficient degree of flexibility.

Due to the border abstraction, the system has an excellent speed advantage over many other procedural and graphical approaches for assembly of larger modules. For example, the sample program shown earlier, which places a ROM in a pad frame, executes in less than two seconds on the VAX 11/780. The ROM has approximately 5000 transistors. The ROM generator produces an eight by eight instance in about 16 seconds. The main limitations are that the SYMBOL data structure consumes about 2KB of memory per symbol and that the routers are not always straightforward to apply due to their somewhat specialized formulation.

Acknowledgements

CFL has taken about two years to develop. I would like to thank the management of the Consortium, in particular, Larry McMurchie, for his encouragement and patience during the critical initial phase of the development. Also, the system would not have achieved its current level of capability without the efforts of several staff members and students who have spent considerable time exploring the package, developing new techniques, new features, and discovering and helping to correct a number of bugs. Most particularly I would like to thank Dave Morgan and Wayne Winder of the Consortium staff, Barry Jinks, Consortium liaison from Microtel Pacific Research, and Jim Schaad, Amilesh Tyagi, and Chyan Yang of the Department of Computer Science here at the University of Washington.

References

1. W. Beckett *Coordinate Free LAP Reference Manual*, UW/NW VLSI Consortium *Design Tools Release 3.0*, University of Washington (June 1985)

2. V. Corbin and B. Yanagida *PLAP Reference Manual*, UW/NW VLSI Consortium *Design Tools Release 2.1*, University of Washington (October 1984)

3. J. Ousterhout, *Editing VLSI Circuits with Ceasar*, 1983 VLSI Tools, Report No. UCB/CSD 83/115 (March 1983)

4. J. Ousterhout, R. N. Mayo, and W. S. Scott, *Magic Tutorials*, Berkeley VLSI Tools, Report No. UCB/CSD 85/225 (March 1985)

Appendix II

MOS Circuit Models in Network C

This paper was submitted to the 1986 Design Automation Conference.

William Beckett
UW/NW VLSI Consortium
Department of Computer Science FR-35
University of Washington
Seattle, Washington 98195

MOS Circuit Models in Network C

Abstract

Network C is a programming language designed for constructing simulation models of VLSI circuits and systems. The language, which is a superset of C, supports a range of modeling capabilities including approximate solution of Kirchoff equations at the circuit level and discrete event functional simulation at the system level. When used to model a MOS circuit, the system first decomposes the circuit into a set of independent stages. The values of nodes, represented by piece-wise linear functions, are communicated between stages using discrete event scheduling. The determination of these piece-wise linear functions is based on continuous time calculations. The result of this hybrid approach is a fast simulation capability which maintains enough accuracy to capture both the digital and analog aspects of a circuit's behavior.

This paper bears on topic 1 (Simulation).

MOS Circuit Models in Network C

Introduction

Network C is a programming language designed for constructing simulation models of VLSI circuits and systems. The language, which is a superset of C, supports a range of modeling capabilities including approximate solution of Kirchoff equations at the circuit level and discrete event functional simulation at the system level. The circuit description capabilities of the language are hierarchical and allow subsystem models of varying levels of precision to be mixed.

In the case of MOS models, the execution of a Network C program has two phases. The first phase is circuit analysis. The effect of circuit analysis is the decomposition of the system being modeled into a set of stages. Each stage is isolated in that the only connections existing between it and all other stages are through nodes connected to gates. After the stages have been isolated in this fashion, the approximate behavior of the system can be obtained by evaluating each stage independently.

The calculation phase of Network C utilizes a combination of continuous time calculation and discrete event scheduling. This technique is aimed at retaining some of the accuracy of purely continuous time systems while realizing the speed advantage inherent in discrete event systems. The value of increased accuracy over purely discrete systems is that a larger class of circuits can be modeled. For example, circuits which utilize analog circuit techniques or in which there is a considerable amount of charge sharing are usually beyond the capabilities of purely switch level simulators. The value of the increased speed of discrete event systems over purely continuous time systems is that circuits with a larger number of components can be accommdated.

Discrete event simulation requires that each stage accept state valued functions as input and produce state valued functions as output. To meet this criterion, Network C models node voltages with piece-wise linear functions. Since piece-wise linear functions can represent an unbounded number of states, Network C reduces the state space by truncating both the derivatives and values of the functions to fixed precission. This produces a set of states which, although it is small relative to a continuous representation, it is still large enough for substantially improved precission when compared to systems with only a few to tens of states.

The continuous time part of the calculation used by Network C differs from the calculation done in SPICE[4] in three fundamental respects. First, since Network C partitions the circuit into stages and computes each stage independently, the rank of the set of node equations is dramatically reduced for larger systems. Second, Network C uses direct three step quadrature rather than a nonlinear equation solver to solve the equations for each stage. Finally, Network C uses only simple DC MOS law models for transistors. All of these aspects tend to trade accuracy for speed.

While Network C may be used to model systems at high levels of abstraction, the purpose of this paper is to provide an overview of the Network C facilities used in developing circuit level models with emphasis on describing the calculation used to evaluate MOS circuits. The first section describes MOS circuit analysis and lists the assumptions about the nature of MOS circuits on which the method is based. The next section discusses behavior calculation in detail. Finally, two short examples are presented. Although a complete definition of the the syntax and semantics of the various Network C constructs is beyond the scope of this paper, many of these constructs are illustrated in the examples.

Circuit Analysis

The MOS abstraction implemented by Network C embodies the following three hypotheses.

1. MOS circuits are composed primarily of gates connected by passive steering networks. The function of the gates is to connect various circuit capacitors to the power and ground rails for charging and discharging. The interpretation of the behavior of the circuit is in terms of the voltages on these capacitors at any point in time. That is, systems are designed so that information is not directly represented by current flows.
2. The power and ground rails have zero impedance and can supply arbitrary currents.
3. The average current into the gate terminals of MOS transistors is zero. Hence, there is no possibility of DC coupling between stages of the circuit.

Assumption 2 is implemented by simply holding the voltages of all device terminals connected to the supply rails constant at the corresponding level.

The Network C uses assumption 3 as the basis for its decomposition of circuits into stages. Each stage consists of a subset of the nodes of the original circuit which may be reached from each other without crossing any gates, that is, by following only source-drain paths. These nodes are called the output nodes of a stage. All devices with either their source or drain connected to an output node in a stage are also considered part of that stage.

Circuit analysis partitions the circuit so that the value of each node in the circuit is determined by exactly one of the stages. That is, all drivers of a node, if there are more than one, belong to the same stage.

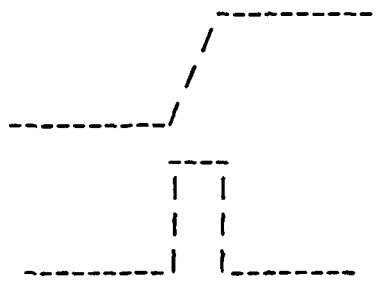
When circuit analysis has completed, each stage will contain zero or more devices whose gates are not connected to nodes determined by that stage. The voltages on these gates are considered to be the independent variables or input nodes from whose values the behavior of the output nodes of stage is computed.

Behavior Calculation

Discrete event scheduling is used to control the operation of the behavior calculation.

The models for each stage, having been derived by circuit analysis, are invoked whenever an input to the stage changes state. The effect of evaluating the model is the calculation of new descriptions for the voltages of each of the output nodes in the stage.

The inputs of a stage are the nodes that are connected to the stage's independent gates. Note that by 'change in state' is meant a change in the parameters of the linear model for a node's voltage, not simply a change in the voltage. To clarify the nature of a change in state, consider the following waveform and its derivative:



This wave is a typical rising edge. It's state changes twice. During the first change, the derivative (which is the slope of the linear model) goes from zero to a positive value, during the next change it goes back to zero. Note that the use of the edges of the derivative as events in this simulator is analogous to use of the edges of the logic level as events in logic level simulators except that, in the case of derivative edges, there are typically two events per logic state change.

The evaluation of each stage involves the determination of a piece-wise linear model for each of its output nodes given piece-wise linear models for each of its input nodes. Although more direct methods are possible for simple stages, currently Network C generates piece-wise linear models by performing continuous time calculations and then fitting the resulting curve with a piece-wise linear form.

The continuous calculations are performed forward in time from the current time point. Since the behaviour of the inputs for future time has not yet been calculated, the calculation of the output forecast is based on the *predicted* behaviour of the inputs.

This prediction of input behaviour is computed as follows. The state of each node in the circuit consists of the three parameters of its linear model, namely:

m	slope
b	intercept
t_0	time of last change

The availability of these parameters means that the value of the node at any future point in time can, in principle, be predicted using the linear formula

$$y = m(t - t_0) + b$$

Actually, the above formula works for large values of $t - t_0$ only if the signal is constant, that is $m = 0$. Otherwise, the formula generates an unbounded value. Therefore, immediately following applications of this formula, Network C bounds the result above and below by the power supply rails. The effect of this heuristic, which works better for CMOS circuits than NMOS circuits, is that the linear models for all nodes in the system are given a piece-wise linear interpretation.

The calculation of outputs from inputs proceeds as follows. For each time point in the forecast range, the system calculates branch currents for each DC branch in the stage. Note that there are no DC branch currents between stages since there are no DC paths between stages. These currents are DC branch currents because, during this part of the computation, all node capacitances are considered to be zero.

The branch currents through each transistor are calculated using the following continuous form the the DC MOS law:

$$\begin{array}{ll} i_{lin} = 2k(v_{gs} - v_{th})v_{ds} - kv_{ds}^2 & v_{gs} - v_{th} \geq v_{ds} \\ i_{sat} = k(v_{gs} - v_{th})^2 & v_{gs} - v_{th} < v_{ds} \\ i_{off} = 0 & v_{gs} - v_{th} < 0 \end{array}$$

noindent The above equations are for NMOS; for PMOS the equations are similar.

Inputs to the calculation are the voltages on the source and drain nodes at time $t - 1$, the forecast time, and the linear model of the gate node. Using the forecast time and the linear model, the gate voltage is determined by the prediction algorithm described above. Outputs from this calculation are the source and drain terminal currents which are the same except for the sign.

After all branch currents for the stage have been computed, the currents for each node are summed. This is similar to the normal Kirchoff procedure except that, since the node capacitances have been disconnected, the result of this summing is a non-zero residual current at the each node.

Next, these residual currents are smoothed using the filter:

$$i = (i_0 + 2i_{t-1} + i_{t-2})/4$$

where i_0 is the unfiltered residual current, i_{t-1} is the filtered residual current at $t - 1$ and i_{t-2} is the filtered residual current at $t - 2$.

The final filtered residual currents are then forced into the node capacitances producing the set of node voltages for this time point. This particular numerical technique is similar to that used in QRS[6].

The above procedure continues until a complete set of output curves for the stage has been computed. Typically this set of curves will span about 50 nano-seconds and contain 50 time points per output node.

Each of these curves is then fit using a piece-wise linear curve fitter. The curve fitter classifies curves by the number of inflections in their second differences and will try to generate a fit having one, two or three line segments. Each segment is the best linear least squares approximation for the points in the segment.

After the curve fitting, a continuity constraint is applied to the resulting piece-wise linear form: in the case in which the new piece-wise linear model for a node intersects the current piece-wise linear model at some future time, the first event in the new piece-wise linear form is delayed until that time.

Finally, the events of the new piece-wise linear model of the node are queued. There can be from one to three of these events and, since the queuing mechanism is preemptive, some events in the new form may preempt events queued earlier.

Since the determination of piece-wise linear models involves a continuous time computation, and since this computation is really a forecast which may have to be recalculated if the assumptions on which it is based change, Network C simulations could potentially require more computation than fixed time step continuous time approaches.

There are several factors that tend to overcome this tendency for increased computation. First, since the scheme is event driven, detailed calculations for any stage are performed only in the neighborhoods of transitions on nodes which are inputs of the stage. Second, when an event occurs on an input that was predicted during a previous invocation of the model of a stage, the model exits immediately and the output is not recalculated. Since the new event was anticipated, its effects have already been included in the output forecast.

Finally, redundant calculations are avoided by using a calculation history. Associated with each stage, these calculation histories consist of a small set of situations and actions. A situation is an encoding of the states of the inputs and outputs of the stage at a time immediately following a change in an input. The action is the set of outputs that were computed in response to the situation. Since the most stages tend not to have very many nodes, and since the parameters of the piece-wise linear models are granular, it is feasible to keep a reasonable number of situations in the history. Also, since the shapes of the transitions for most nodes in digital circuits are determined by time invariant physical characteristics, it is likely that the situations recur. When this happens, the output for the stage is available immediately as a result of table look up in the calculation history.

The net effect of all these mechanisms is that the resolution of the simulator is variable. In the worst case of continuously changing input, say a sinusoid, the forecast and fit procedures will generate a large number of short lived piece-wise linear models of localities of the sine wave. Of course, using this mechanism to track a sine wave is expensive. Generally speaking, the more the natural behaviour of the nodes of the circuit fit the assumptions of the forecasts the less the amount of calculation required. In the best case it is possible for the simulator to completely learn the circuit and, once that happens, all behaviour is obtained by table lookup.

Examples

In discussing the examples, a number of the features of Network C language and translator will be highlighted. The Network C translator is implemented using a three pass preprocessor. The first pass uses a *yacc* parser to build a complete parse tree. The grammar used by the parser consists of the grammar for the portable version of the C compiler with the extra Network C constructs added. The second pass applies a number of parse tree rewriting rules to convert the subtrees containing Network C constructs into subtrees containing only C constructs. The third pass walks the modified parse tree outputing the text of the C translation of the original Network C input.

The C programs generated by the Network C translator can be compiled by the system C compiler and the resulting object files represent executable models of VLSI subsystems. These models, like all C procedures, may be put on libraries for inclusion in other, more complex models.

Network C programs have two module types - network descriptions (called circuits) and procedural device or subsystem models (called models). Circuits, or network descriptions, consist of a set of elements connected to a set of nodes. An element is either another circuit or a model.

Models are procedures which compute the values of outputs from inputs.

The first example, shown in Figure 1, is a simple series of three CMOS inverters. The example illustrates the general form of Network C programs.

```
maincircuit mos9()
/*  Simplified cmos inverter chain              */
{
      elements
      {
         g1    (gen, 0.0,5.0,1.0e9,-1.0e9)  clk, a, x;
         i1    (cinv,5.0,0.0,1.0e-4,1.0e-4) a,   b;
         i2    (cinv,5.0,0.0,1.0e-4,1.0e-4) b,   c;
         i3    (cinv,5.0,0.0,1.0e-4,1.0e-4) c,   d;
      };
      conditions
      {
          clk = (clk+1) % 2; [50.e-9]
      };
}
```

Figure 1.

The declarator *elements* introduces the list of network elements. Each network element

has an instance name, a class name, a set of optional parameters, and a terminal connection list.

The first element in this circuit, *g1*, is a clock generator which generates a clock signal *a*. *a* is synchronized to the signal *clk* and has a minimum value of 0.0, a maximum value of 5.0, and rise and fall rates of 1 volt per nanosecond. The second phase of the clock, *x*, is not used in this experiment.

The next three elements form the series of inverters. The clock, *a*, drives the first inverter. Its output, *b*, drives the next inverter whose output, *c*, drives the last inverter. The parameters following the class name, *cinv*, initialize instance variables within the definition of *cinv* (see below).

The logical clock, *clk*, is generated by the statement following the *conditions* declarator. The interpretation of this statement is that *clk* will oscillate between 0 and 1 with a transition occurring every 50 nanoseconds.

The procedure *cinv* shown in Figure 2 computes the input/output relationships of the inverter. The declaration

network float trigger a;

is a Network C construct that specifies that the first terminal of the inverter will be connected to a node which has a floating point value. (Network C also allows integer nodes.) The *trigger* specification indicates that the model is to be invoked every time this quantity changes state. Recall that the state is defined to be the state of the piece-wise linear model of the quantity so *cinv* will get control every time the node connected to its first terminal changes either its slope or its intercept.

The declaration

network float mos y;

indicates that the second terminal of the inverter is also connected to a floating point node. This terminal is not a *trigger* which means that the inverter will not get control if the node it is connected to changes. Also, the specification *mos* means that the node connected to this terminal is to be considered part of a MOS circuit and MOS analysis will result. This specification is required since Network C allows other types of network nodes (for example, *clk* above) which are not considered by MOS analysis. In the case of this circuit, circuit analysis will put each instance of *cinv* in a different stage.

The general rule is that the nodes appearing in the terminal list of an element in a *elements* declaration are bound in order to the *network* variables declared in the model. Similarly, the quantities in the parameter list of an element are taken in order to initialize *local* variables in the model. For example, all the instances of *cinv* in Figure 1 have the same set of parameters. They are, *vdd* (set to 5.0), *gnd* (set to 0.0), *kpu* and *kpd* (both set to 1.0e-4).

```
#define  clip(v,vl,vh)  amin1(amax1((vl),(v)),(vh))

cinv(tf,vout,iout)
float  tf;                    /* forecast time                          */
float  vout,                  /* output voltage                         */
float *iout;                  /* output current, returned               */

/*  CMOS inverter                                                        */
{
      network float trigger a;
      network float mos     y;

      local float vdd,        /* high power supply voltage              */
                  gnd,        /* low  power supply voltage              */
                  kpu,        /* k for the pull up   transistor         */
                  kpd;        /* k for the pull down transistor         */
      float       vtn,vtp,    /* threshold voltages                     */
                  v1,         /* input voltage                          */
                  ipd,ipu;    /* transistor currents                    */

/*  Threshold voltages are 0.2*Vdd.                                      */

      vtn =  0.2*vdd;    vtp = -0.2*vdd;

/*  Generate the input forecast and clip.                                */

      v1 = a'*(clock + tf - a->t0) + a;
      v1 = clip(v1,gnd,vdd);

/*  Compute transistor currents.                                         */

      pmos0(v1, vdd, vout, kpu, vtp, &ipu);
      nmos0(v1, gnd, vout, kpd, vtn, &ipd);

/*  Compute output current.                                              */

      *iout = ipd + ipu;
}
```

Figure 2.

The arguments to *cinv* are the forecast time and the output voltage. The output will be the output current.

The thresholds for the *n* and *p* device are computed from the power supply and the gate voltage is obtained using the linear prediction. Transistor currents are computed using the DC MOS law and summed to produce the output.

nmos0 and *pmos0* implement the continuous MOS law stated earlier for *n* and *p* transistors respectively.

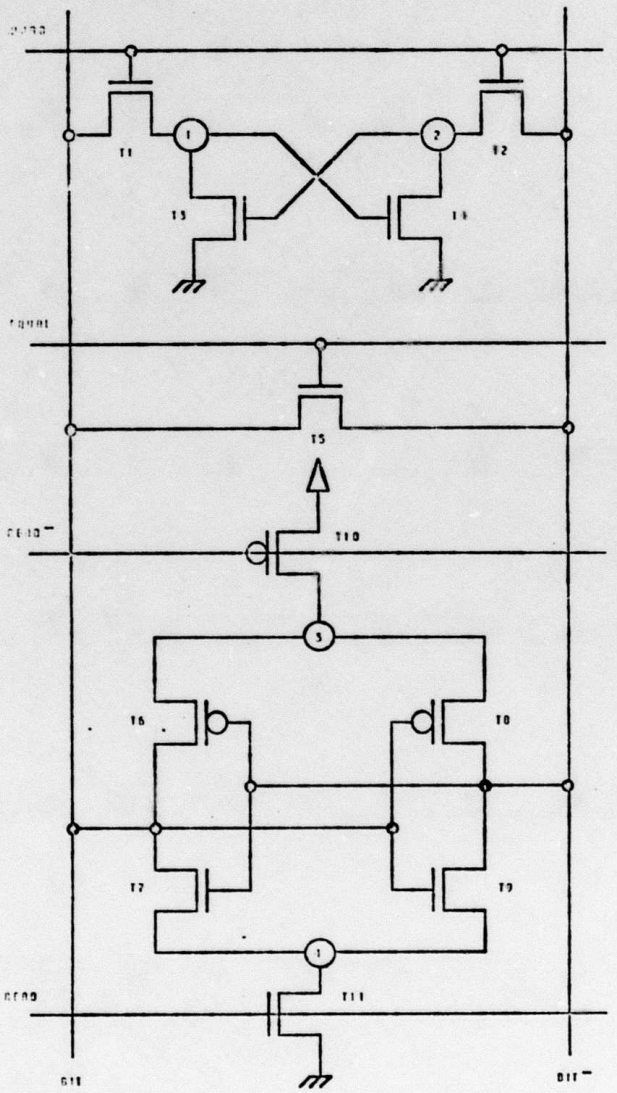


Figure 3.

The next example is the CMOS RAM cell and sense amplifier shown in Figure 3.

Since this circuit is closely coupled, circuit analysis will place all nodes and devices in the same stage. The circuit is a good example of a circuit whose behavior calculation requires the modeling of both analog and digital characteristics.

```
maincircuit ram()
{
/*  RAM cell and sense amp                                              */

      nodecap
      {
         bit = 1600.e-15;
         bitn =1600.e-15;
         n01  = 160.e-15;
         n02  = 160.e-15;
      };
      elements
      {
         g1   (gen, 0.0,5.0,1.0e9,-1.0e9)   eq,    equal, equaln
         g2   (gen, 0.0,5.0,1.0e9,-1.0e9)   clk,   read,  readn
         g3   (gen, 0.0,5.0,1.0e9,-1.0e9)   clk,   word,  wordn

         t01  (nmosfet, 2.33e-5,  1.0)      word,  bit,   n01
         t02  (nmosfet, 2.33e-5,  1.0)      word,  bitn,  n02
         t03  (nmosfet, 2.33e-5,  1.0)      n02,   n01,   gnd
         t04  (nmosfet, 2.33e-5,  1.0)      n01,   n02,   gnd

         t05  (nmosfet, 6.99e-5,  1.0)      equal, bit,   bitn

         t06  (pmosfet, 6.99e-5, -1.0)      bitn,  n03,   bit
         t07  (nmosfet, 4.66e-5,  1.0)      bitn,  bit,   n04

         t08  (pmosfet, 6.99e-5, -1.0)      bit,   n03,   bitn
         t09  (nmosfet, 4.66e-5,  1.0)      bit,   bitn,  n04

         t10  (pmosfet, 6.99e-5, -1.0)      readn, vdd,   n03
         t11  (nmosfet, 4.66e-5,  1.0)      read,  n04,   gnd
       };
}
```

Figure 4.

The circuit description given in Figure 4 is similar in form to that of the three inverters of the first example. In this case, the components consist of three waveform generators and the *n* and *p* transistors modeled by *nmosfet* and *pmosfet* respectively.

The *nodecap* declarator is used to associate specific capacitance values with nodes. (There is a small default minimum node capacitance associated with all MOS nodes which is required by the numerical method.) In this case, the *bit* and *bitn* lines are given large capacitances, 1600 femtofarads, and the memory cell storage nodes are given smaller, but larger than typical, values.

nmosfet uses the same basic DC MOS law model described above to calculate drain-source current from terminal voltages. For the *nmosfet* and *pmosfet* models, however, the gain supplied as the first parameter is one half the device transconductance, that is

$$gain = k/2.$$

The device conductance, k, is defined as the process conductance times the width over the length, that is,

$$k = k'(w/l)$$

where k' is the process conductance given by

$$k' = \mu c_{ox}.$$

μ is the carrier mobility and c_{ox} is the gate oxide capacitance per unit area.

If ϵ_{ox} is the permittivity and t_{ox} is the thickness of the gate dielectric, then

$$c_{ox} = \epsilon_{ox}/t_{ox}.$$

For this example we have taken

$\mu = 600$	$cm^2/volt\text{-}sec$
$\epsilon_{ox} = 3.9\epsilon_0 = 3.5 \times 10^{-13}$	$farads/cm$
$t_{ox} = 0.1 \times 10^{-6}$	$meters$ (1000 $angstroms$).

Therefore,

$c_{ox} = 35.0 \times 10^{-9}$	$farads/cm^2$
$k' = \mu c_{ox} = 2.0 \times 10^{-5}$	$amps/volt^2$ (approx)

and the gain parameters for the transistor models are given by

$$gain = k/2 = k'w/2l = 1.0 \times 10^{-5}(w/l).$$

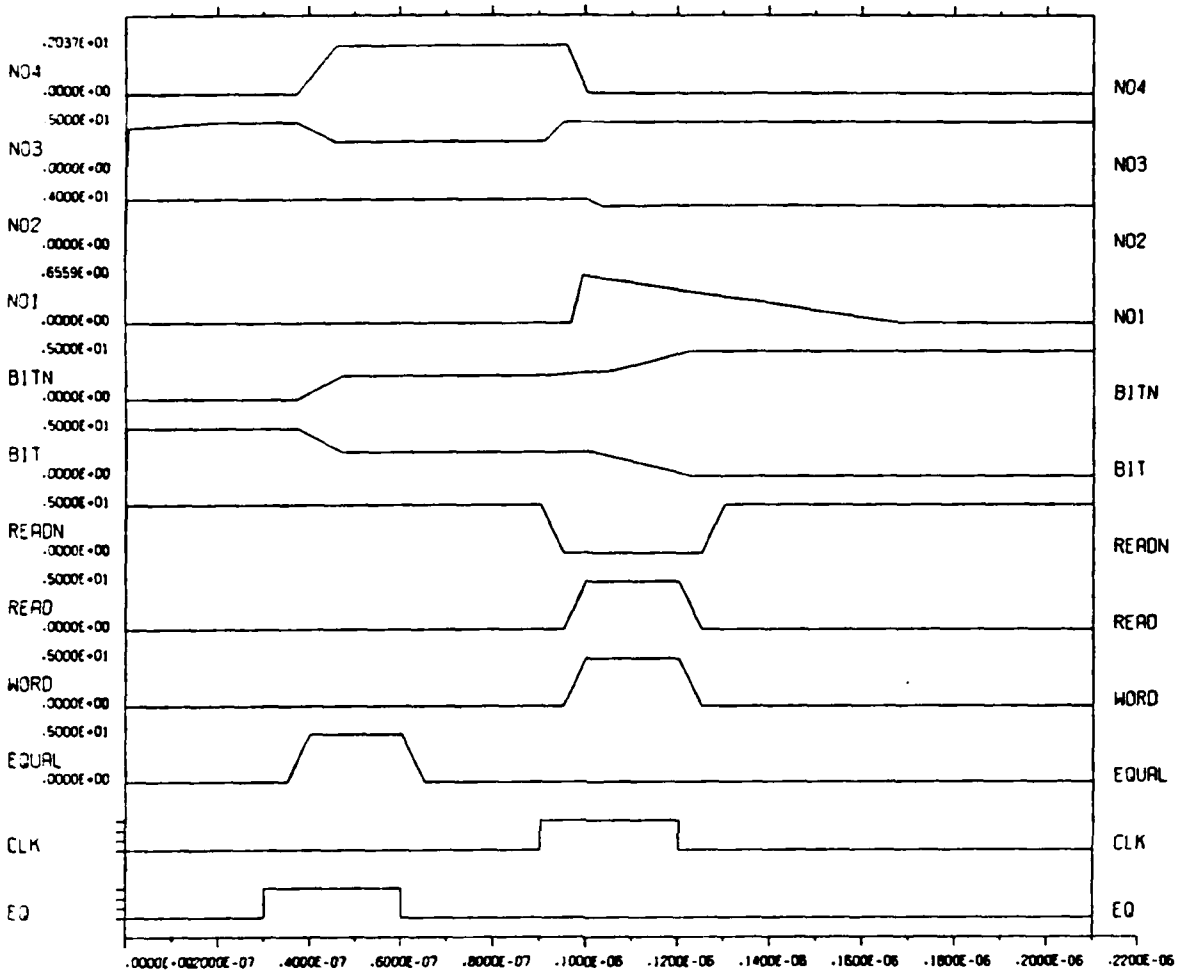


Figure 5.

The *vt* parameters are 1.0 for *n* transistors and -1.0 for *p* transistors.

Referring to the circuit diagram for the RAM cell and sense amplifier shown in figure 3, the memory cell itself consists of the four transistors, T1, T2, T3, and T4. The information stored on the gates of T3 and T4 is made available on *bit* and *bitn* when the *word* line is brought up.

Prior to bringing up the word line, *bit* and *bitn* are brought to the same potential, or equalized, by raising the *equal* line which turns on T5.

The sense amplifier, T6, T7, T8, T9, T10, and T11, consists of a pair of cross coupled inverters connected to *bit* and *bitn*. When the *word* line and *read* line are brought up

together, the amplifier accelerates the transition of the bit lines and restores the cell.

The behavior for this cell calculated by the Network C model is shown in Figure 5. In looking at this plot it is important to note that the waveforms are not all drawn using the same vertical scale. *clk* and *eq* are control signals for the waveform generators.

In this experiment, *bit* is started at 5.0 volts and *bitn* is started at 0.0 volts but *n01* is started at 0.0 volts and *n02* is started at 4.0 volts. Therefore, the state of the memory cell is the opposite of the state of the bit lines.

When the *equal* line is brought up, *bit* and *bitn* both make transitions to 2.5 volts. When the *word* and *read* lines are brought up, the state of the memory cell, represented by the charge on nodes *n01* and *n02*, is quickly transferred to the bit lines. Note that, in the process, the state of the memory cell is restored. *n02* has dropped slightly and *n01* has returned to 0.0. Note that *n01* did rise to 0.6 volts during the read when it was suddenly connected to the highly charged *bit* line.

Acknowledgements

Network C is based on work done originally as part of the author's dissertation. Because the development has been spread over a number of years, many more people have contributed than can be reasonalby mentioned here. Special thanks must be given to my chairman, Bob Herriot, and to Neil Runstein of the Academic Computer Center, both of whom have now left the University of Washington.

More recently, Rob Daasch, formerly of the VLSI Consortium, helped considerably during the development of the MOS calculation presented in this paper. Wayne Winder of the VLSI Consortium has helped in converting the system from the CYBER 855 to Unix. Also, I would like to thank Barry Jinks, VLSI Consortium liaison from Microtel Pacific Research, for his RAM cell and sense amp design which was used as the last example. Finally, I would like to thank the VLSI Consortium management, particularly Larry McMurchie, for supporting the development of this system in our current environment.

References

1. W. Beckett. *A Hybrid Paradigm for Computer Programming and Its Investigation in the Context of Electronic Circuit Simulation by Means of an Extensible Language*, Ph.D. Dissertation, Technical Report No. 78-05-02, Department of Computer Science, University of Washington (1978).

2. E. Lelarasmee and A. Sangiovanni-Vincentelli, *RELAX: A New Circuit Simulator for Large Scale MOS Integrated Circuits*, ACM IEEE 19th Design Automation Conference Proceedings (June 1982).

3. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Massachusetts (1980).

4. L. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, ERL Memo No. ERL-M520, University of California, Berkely (1975).

5. C. J. Terman, *Simulation Tools for Digital LSI Design*, Ph.D. Dissertation, Laboratory for Computer Science, Massachusetts Institute of Technology (1983).

6. Vivid Reference Manual, Microelectronics Center of North Carolina (1983).

Appendix III

University of Washington

Abstract

A Variable Digital Filter Design in 3um CMOS

by Hyong Yong Lee

Chairperson of the Supervisory Committee: Professor Mani Soma
Department of Electrical Engineering

Variable Infinite Impulse Response(IIR) Filter has been designed in a parallel form to be used as a filter bank for preliminary speech processing. It can realize up to 8th order IIR filter with accuracy due to its 16-bit word-length and architecture. The simulation shows it can sample up to 16 Khz. The 25,000 transistors occupy a circuit area of 7.9mm by 9.2mm and it is packaged in a 64 pin Dual In-line Package (DIP). It has been designed using 3 μm Complementary Metal Oxide Semiconductor (CMOS) P-well technology supported by Metal Oxide Semiconductor Implementation Service (MOSIS). The testing has been performed using an IBM-PC AT† with the Northwest Instrument Systems μAnalyst 2000.

Considerable design time and simplification of the control resulted from the usage of the shift register array (SRA) rather than Random Access Memory (RAM) to store the filter coefficients, data and intermediate results. This was possible due to usage of Master Slave Flip Flop (MSFF) not only as basic cells for SRA but for 9-bit counter, 32-bit accumulator and two 16-bit latches. The 2-to-1 selector, counter reset circuitry and differently sized buffers were the only other cells designed.

† IBM-PC AT is a trademark of International Business Machine.